42P15452 *Patent*

UNITED STATES PATENT APPLICATION

FOR

INCREMENTAL CHECKPOINTING IN A MULTI-THREADED ARCHITECTURE

INVENTORS:

SHUBHENDU S. MUKHERJEE
STEVEN K. REINHARDT
JOEL S. EMER

PREPARED BY:

BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN, LLP
12400 WILSHIRE BOULEVARD
SEVENTH FLOOR
LOS ANGELES, CA 90025-1026

(503) 684-6200

EXPRESS MAIL NO. EV 325528406 US

INCREMENTAL CHECKPOINTING IN A MULTI-THREADED ARCHITECTURE

RELATED APPLICATIONS

This U.S. Patent application is related to the following U.S. Patent applications:

(1) HARDWARE RECOVERY IN A MULTI-THREADED ARCHITECTURE, application number (Attorney Docket No. P15451), filed August 29, 2003; and

(2) PERIODIC CHECKPOINTING IN A REDUNDANTLY MULTI-THREADED ARCHITECTURE, application number (Attorney Docket No. P15453), filed August 29, 2003.

TECHNICAL FIELD

[0001] The invention relates to multi-threaded computing architectures. More specifically, the invention relates to devices and techniques for incremental, periodic storage of checkpoints in a multi-threaded processor.

BACKGROUND

[0002] Processors are becoming increasingly vulnerable to transient faults caused by alpha particle and cosmic ray strikes. These faults may lead to operational errors referred to as "soft" errors because these errors do not result in permanent malfunction of the processor. Strikes by cosmic ray particles, such as neutrons, are particularly critical because of the absence of practical protection for the processor. Transient faults currently account for over 90% of faults in processor-based devices.

[0003] As transistors shrink in size the individual transistors become less vulnerable to cosmic ray strikes. However, decreasing voltage levels the accompany the decreasing transistor size and the corresponding increase in transistor count for the processor results in an exponential increase in overall processor susceptibility to cosmic ray strikes or other causes of soft errors. To compound the problem, achieving a selected failure rate for a multi-processor system requires an even lower failure rate for the individual processors. As a result of these trends, fault detection and recovery techniques, typically reserved for mission-critical applications, are becoming increasing applicable to other processor applications.

[0004] Several terms are commonly used when discussing processor errors and error recovery. A Failure in Time (FIT) refers to an error rate of one failure in one billion (10^9) hours. Mean Time Between Failure (MTBF) is the time between failures caused by soft errors. MTBF requirements are typically expressed in years. For example, one FIT equals a MTBF of 114,155 years: $114{,}155 = \frac{10^9}{(24 * 365)}$.

[0005] Silent Data Corruption (SDC) occurs when errors are not detected and may result in corrupted data values that can persist until the processor is reset. The SDC Rate is the rate at which SDC events occur. Soft errors are errors that are detected, for example, by using parity checking, but cannot be corrected. The rate of these detected, unrecoverable errors is referred to as the DUE rate.

[0006] For example, publicly available documents from IBM (D.C. Bossen, "CMOS Soft Errors and Server Design," IBM Server Group, Reliability Physics Tutorial Notes, Reliability Fundamentals, April 2002.), specify 25 years MTBF for DUE and 1000 years MTBF for SDC. These specifications are for single-processor systems. Application to a

multi-processor system results in more stringent specifications for individual processors. It is becoming increasingly difficult to meet SDC and DUE FIT specifications because the neutron FIT contribution of latches is increasing. Other components, for example, most SRAM cells, either can be protected via interleaved parity or error correcting codes or do not provide significant contribution to the overall FIT rate.

[0007] The FIT rate of latches consists of two parts: the raw FIT rate and a derating factor. The raw FIT rate can be computed using circuit models and currently ranges between 0.001 and 0.01 per latch. The derating factor is the fraction of faults that lead to errors. Typically, the derating factor is 10%. See, for example, Eugene Normand, "Single Event Upset at Ground Level," IEEE Transactions on Nuclear Science, Vol. 43, No. 6, December 1996 and Y. Tosaka, et al., "Impact of Cosmic Ray Neutron Induced Soft Errors on Advanced Submicron CMOS Circuits," VLSI Symposium on VLSI Technology Digest of Technical Papers, 1996. Using the specifications set forth above as a further example, in a 64-processor system, each processor can have only approximately 1,800 latches. However, designing a complex, high-performance processor core with only 1,800 latches is extremely difficult.

[0008] Fault detection support can reduce a processor's SDC rate by halting computation before faults can propagate to permanent storage. Parity, for example, is a well-known fault detection mechanism that avoids silent data corruption for single-bit errors in memory structures. Unfortunately, adding parity to latches or logic in high-performance processors can adversely affect the cycle time and overall performance. Consequently, processor designers have resorted to redundant execution mechanisms to detect faults in processors.

[0009] Current redundant-execution systems commonly employ a technique known as "lockstepping" that detects processor faults by running identical copies of the same program on two identical lockstepped (cycle-synchronized) processors. In each cycle, both processors are fed identical inputs and a checker circuit compares the outputs. On an output mismatch, the checker flags an error and can initiate a recovery sequence. Lockstepping can reduce a processors SDC FIT by detecting each fault that manifests at the checker. Unfortunately, lockstepping wastes processor resources that could otherwise be used to improve performance.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention is illustrated by way of example, and not by way of limitation, in the figures of the accompanying drawings in which like reference numerals refer to similar elements.

Figure 1 is a block diagram of one embodiment of a redundantly multithreaded architecture with the redundant threads.

Figure 2 is a block diagram of one embodiment of a simultaneous and redundantly threaded architecture.

Figure 3 illustrates minimum and maximum slack relationships for one embodiment of a simultaneous and redundantly multithreaded architecture.

Figure 4 illustrates a pipelined relationship between a branch instruction and a branch target instruction.

Figure 5 illustrates minimum and maximum slack relationships for one embodiment of a simultaneous and redundantly multithreaded architecture with recovery.

Figure 6 is a conceptual illustration of one embodiment of a history buffer that can be used for incremental checkpointing.

Figure 7 is a flow diagram of one embodiment of error recovery using a history buffer.

Figure 8 is a block diagram of an electronic system that can provide an environment for multithreaded processors.

DETAILED DESCRIPTION

[0010] Methods and apparatuses for incremental, periodic storage of checkpoints in a multi-threaded processor are described. In the following description, for purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the invention. It will be apparent, however, to one skilled in the art that the invention can be practiced without these specific details. In other instances, structures and devices are shown in block diagram form in order to avoid obscuring the invention.

Sphere of Replication

[0011] **Figure 1** is a block diagram of one embodiment of a redundantly multithreaded architecture. In a redundantly multithreaded architecture faults can be detected by executing two copies of a program as separate threads. Each thread is provided with identical inputs and the outputs are compared to determined whether an error has occurred. Redundant multithreading can be described with respect to a concept referred to herein as the "sphere of replication." The sphere of replication is the boundary of logically or physically redundant operation.

[0012] Components within sphere of replication 130 (e.g., a processor executing leading thread 110 and a processor executing trailing thread 120) are subject to redundant execution. In contrast, components outside sphere of replication 130 (e.g., memory 150, RAID 160) are not subject to redundant execution. Fault protection is provide by other techniques, for example, error correcting code for memory 150 and parity for RAID 160. Other devices can be outside of sphere of replication 130 and/or other techniques can be used to provide fault protection for devices outside of sphere of replication 130.

[0013] Data entering sphere of replication 130 enter through input replication agent 170 that replicates the data and sends a copy of the data to leading thread 110 and to trailing thread 120. Similarly, data exiting sphere of replication 130 exit through output comparison agent 180 that compares the data and determines whether an error has occurred. Varying the boundary of sphere of replication 130 results in a performance versus amount of hardware tradeoff. For example, replicating memory 150 would allow faster access to memory by avoiding output comparison of store instructions, but would increase system cost by doubling the amount of memory in the system.

[0014] In general, there are two spheres of replication, which can be referred to as "SoR-register" and "SoR-cache." In the SoR-register architecture, the register file and caches are outside the sphere of replication. Outputs from the SoR-register sphere of replication include register writes and store address and data, which are compared for faults. In the SoR-cache architecture, the instruction and data caches are outside the sphere of replication, so all store addresses and data, but not register writes, are compared for faults.

[0015] The SoR-cache architecture has the advantage that only stores (and possibly a limited number of other selected instructions) are compared for faults, which reduces checker bandwidth and improves performance by not delaying the store operations. In contrast, the SoR-register architecture requires comparing most instructions for faults, which requires greater checker bandwidth and can delay store operations until the checker determines that all instructions prior to the store operation are fault-free. The SoR-cache can provide the same level of transient fault coverage as SoR-register because

faults that do not manifest as errors at the boundary of the sphere of replication do not corrupt the system state, and therefore, are effectively masked.

[0016] In order to provide fault recovery, each instruction result should be compared to provide a checkpoint corresponding to every instruction. Accordingly, the SoR-register architecture is described in greater detail herein.

<u>Overview of Simultaneous and Redundantly Threaded Architecture</u>

[0017] **Figure 2** is a block diagram of one embodiment of a simultaneous and redundantly threaded architecture. The architecture of Figure 2 is a SoR-register architecture in which the output, or result, from each instruction is compared to detect errors.

[0018] Leading thread 210 and trailing thread 220 represent corresponding threads that are executed with a time differential so that leading thread 210 executes instructions before trailing thread 220 executes the same instruction. In one embodiment, leading thread 210 and trailing thread 220 are identical. Alternatively, leading thread 210 and/or trailing thread 220 can include control or other information that is not included in the counterpart thread. Leading thread 210 and trailing thread 220 can be executed by the same processor or leading thread 210 and trailing thread 220 can be executed by different processors.

[0019] Instruction addresses are passed from leading thread 210 to trailing thread 220 via instruction replication queue 230. Passing the instructions through instruction replication queue 230 allows control over the time differential or "slack" between execution of an instruction in leading thread 210 and execution of the same instruction in trailing thread 220.

[0020] Input data are passed from leading thread 210 to trailing thread 220 through source register value queue 240. In one embodiment, source register value queue 240 replicates input data for both leading thread 210 and trailing thread 220. Output data are passed from trailing thread 220 to leading thread 210 through destination register value queue 250. In one embodiment, destination register value queue 240 compares output data from both leading thread 210 and trailing thread 220.

[0021] In one embodiment, leading thread 210 runs hundreds of instructions ahead of trailing thread 220. Any number of instructions of "slack" can be used. In one embodiment, the slack is caused by slowing and/or delaying the instruction fetch of trailing thread 220. In an alternate embodiment, the slack can be caused by instruction replication queue 230 or an instruction replication mechanism, if instruction replication is not performed by instruction replication queue 230.

[0022] Further details for techniques for causing slack in a simultaneous and redundantly threaded architecture can be found in "Detailed Design and Evaluation of Redundant Multithreading Alternatives," by Shubhendu S. Mukherjee, Michael Kontz and Steven K. Reinhardt in *Proc. 29th Int'l Symp. on Computer Architecture*, May 2002 and in "Transient Fault Detection via Simultaneous Multithreading," by Steven K. Reinhardt and Shubhendu S. Mukherjee, in *Proc. 27th Int'l Symp. on Computer Architecture*, June 2000.

[0023] **Figure 3** illustrates minimum and maximum slack relationships for one embodiment of a simultaneous and redundantly threaded architecture. The embodiment of Figure 3 is a SoR-register architecture as described above. The minimum slack is the total latency of a cache miss, latency from execute to retire, and latency incurred to

forward the load address and value to the trailing thread. If the leading thread suffers a cache miss and the corresponding load from the trailing thread arrives at the execution point before the minimum slack, the trailing thread is stalled.

[0024] Similarly, the maximum slack is latency from retire to fault detection in the leading thread. In general, there is a certain amount of buffering to allow retired instructions from the leading thread to remain in the processor after retirement. This defines the maximum slack between the leading and trailing threads. If the buffer fills, the leading thread is stalled to allow the trailing thread to consume additional instructions from the buffer. Thus, if the slack between the two threads is greater than the maximum slack, the overall performance is degraded.

Overview of Checkpointing and Backward Recovery

[0025] One fault-recovery technique, referred to as "backward recovery" involves restoring a system (e.g., a processor, a computer system) to an earlier fault-free state and re-executing a thread from the restoration point. A copy of the earlier state that can be restored is referred to as a "checkpoint." Backward recovery includes two issues to be addressed to provide transparent recovery: non-deterministic events and regenerated outputs.

[0026] If a non-deterministic event (e.g., an asynchronous interrupt) occurs after the last checkpoint re-execution after a fault may not follow the same path as the original execution. Specifically, if an externally visible output (e.g., a store) was generated along the original execution path prior to the fault, but the re-execution follows a different path that generates a different output, the resulting sequence of outputs, as observed from a reference external to the system, will not be consistent with fault-free execution and

recovery will not be transparent. To prevent this occurrence, a backward recovery system must guarantee at each output operation that any subsequent fault-induced rollback will follow the same execution path up to the point of the output. This is referred to as the "output commit" problem.

[0027] Even if the re-execution deterministically follows the same path as the original execution, any externally visible output operations after the last checkpoint will be performed again during re-execution. If these output operations are not idempotent, then the re-execution will lead to behavior inconsistent with a fault-free execution and recovery will not be transparent.

[0028] One solution to both issues (non-deterministic events and regenerated outputs) is to create a new checkpoint automatically with each output operation. The checkpoint then incorporates any non-deterministic events that may have led to the execution of that output operation. If the output operation completes successfully, then any subsequent fault will roll back to the execution point after that operation and the output operation will not be re-executed.

[0029] Deeply-pipelines processors typically predict branch directions and addresses to support speculative execution of instructions after the branch. As illustrated in **Figure 4**, the outcome of the branch instruction is not determined until the instruction has been executed. Consequently, in the absence of branch prediction, instructions following the branch may be delayed. This is one reason that deeply-pipelined processors support aggressive branch prediction and recovery mechanisms. On branch misprediction, the processor can recover the processor state at the branch instruction and restart the pipelined instructions.

[0030] One solution to provide checkpointing is to store the complete architectural register file at each branch instruction. This solution could be expensive in terms of increased hardware requirements and performance loss. Consequently, processors can use incremental checkpointing in which for each instruction the prior value of a register is stored before a new value is written to the register. Then, to recover from a branch misprediction, the processor restores the most recent register values from the table.

[0031] This incremental checkpoint can be further optimized. In dynamically-scheduled processors that include more registers than are supported in the instruction set architecture, these processors create a mapping form the architectural register space (as supported in the instruction set) to the physical register space (as supported by the physical implementation). The destination register of an instruction is mapped to a physical register. When a new physical mapping is created for an architectural register, the previous mapping of the register is stored (e.g., in a table). To recover from a branch, the correct register mapping is restored at the point of the branch. Restoring the mapping is simpler than restoring the register values because the number of map bits required is typically smaller than, for example, 64-bit register values.

<u>Overview of Simultaneous and Redundantly Threaded Architecture with Recovery</u>

[0032] The simultaneous and redundantly threaded architecture with recovery (SRTR) architecture extends the simultaneous and redundantly threaded (SRT) architecture to support hardware recovery. The SRTR architecture provides recovery support through use of periodic checkpointing mechanisms
for recovery purposes. A deeply-pipelined processor can include a checkpointing mechanism to recover from branch mispredictions. A processor having a SRTR

architecture can use a modified checkpointing technique to recover from transient hardware faults.

[0033] In order to provide recovery from transient hardware faults the processor architecture performs fault detection prior to the instruction's retire point. The fault detection prior to the retire point is caused by the processor freeing the checkpoint corresponding to an instruction when the instruction is retired.

[0034] **Figure 5** illustrates minimum and maximum slack relationships for one embodiment of a simultaneous and redundantly multithreaded architecture with recovery. Providing fault detection prior to the time at which an instruction is retired reduces the maximum allowed slack between the leading thread and the trailing thread as compared to the SRT architecture. In the SRT architecture, load addresses and values are forwarded from the leading thread to the trailing thread after the load instruction is retired. Because the SRTR architecture performs fault detection prior to retirement of the load instruction, the leading thread forwards load addresses and values to the trailing thread after the load execution is complete, but before the load instruction is retired. Consequently, the source register value queue (as illustrated in Figure 2) is not a first-in/first-out (FIFO) queue, but is a speculative structure that can recover from branch mispredictions.

[0035] The SRTR architecture has a lower maximum slack compared to the SRT architecture because the maximum slack is the latency between the completion and fault detection point of an instruction. Increasing this latency is a more complex design issue than increasing the maximum slack in the SRT architecture. This is because instructions that have been executed but have not yet been retired cannot commit the results to the

destination register in the architectural register file until the instructions have passed the fault detection and retirement stages.

One Embodiment of a Technique to Use a History Buffer for Incremental Checkpointing

[0036] **Figure 6** is a conceptual illustration of one embodiment of a history buffer that can be used for incremental checkpointing. In one embodiment, the history buffer is a table with multiple entries. Each entry includes information related to a retired instruction, for example, the instruction pointer, the old destination register value and the physical register to which the destination architecture register is mapped. Other and/or different data can be stored in the history buffer.

[0037] The instruction pointer can be either the program counter or an implementation-dependent instruction number. The old destination value records the value of the register before the associated instruction rewrote the register with a new value. The register map identifies which physical register was updated. The number of entries in the history buffer is dependent on the implementation and, in one embodiment, is chosen to avoid pipeline stalls; however, other factors can be considered when implementing the history buffer. Only one history buffer is required for a pair of redundant threads.

[0038] When an instruction is retired and before the results are written to the destination register in the architecture register file, an entry is created in the history buffer with, in one embodiment, the instruction pointer, old destination register file and register map. Reading the old register value from the architecture register file may require an additional read port in the register file. By updating the history buffer when an

instruction is retired an incremental checkpoint corresponding the processor state at the time the instruction is retired is created.

[0039] When a retired instruction passes the fault detection stage, the corresponding entry in the history buffer can be freed. Passing the fault detection stage indicates that execution of the instruction has been completed fault-free and the corresponding checkpoint is no longer needed. During the period when data is stored in the history buffer, the entries of the history buffer are used to restore the architectural state corresponding to a checkpoint.

Overview of Recovery Using the History Buffer

[0040] **Figure 7** is a flow diagram of one embodiment of error recovery using a history buffer. Fault detection operates as described above by comparing output data from the leading thread and the trailing thread and storing data in the history buffer. When no error is detected at 700 both threads continue to execute. When an error is detected at 700 error recovery using the history buffer is triggered.

[0041] When the fault detection mechanism detects an error in a retired instruction, the recovery mechanism causes both threads to flush all speculative instructions that have not been retired, 710. The architectural state of the trailing thread is also flushed, 720. In one embodiment, flushing the architectural state of the trailing thread includes flushing the architectural register file of the trailing thread.

[0042] The architectural register file of the leading thread is reconstructed, 730. In one embodiment, the contents of the history buffer and the contents of the register file are used to reconstruct the architectural file corresponding to the preceding checkpoint. The architectural register file includes correct values up to the last retired instruction.

[0043] The values are rolled back to the instruction for which the fault was detected. In one embodiment, this is accomplished by finding the oldest update to each register from the history buffer. The history buffer is flushed, 740, after the architectural register file is reconstructed.

[0044] The contents of the leading thread architectural register file are loaded to the architectural register file of the trailing thread, 750. Both threads are restarted at the instruction that caused the fault, 760.

[0045] Logically, the history buffer is outside of the sphere of replication. In one embodiment, all instructions entering the history buffer are compared for mismatch. This can be accomplished via the fault detection mechanism. In one embodiment, the history buffer is protected with ECC to allow recovery on a single strike to the history buffer. In one embodiment, all data paths and logic from the fault detection module to the history buffer are protected, for example, via ECC or dual-rail logic.

Example of a System

[0046] **Figure 8** is a block diagram of an electronic system that can provide an environment for multithreaded processors. The electronic system illustrated in Figure 8 is intended to represent a range of electronic systems. Alternative electronic systems can include more, fewer and/or different components.

[0047] Electronic system 800 includes bus 810 or other communication device to communicate information, and processor(s) 820 coupled to bus 810 to process information. Electronic system 800 further includes random access memory (RAM) or other dynamic memory as well as static memory, for example, a hard disk or other storage device 835 (referred to as memory), coupled to bus 810 via memory controller 830 to store information

and instructions to be executed by processor(s) 820. Memory 835 also can be used to store temporary variables or other intermediate information during execution of instructions by processor(s) 820. Memory controller 830 can include one or more components to control one or more types of memory and/or associated memory devices. Electronic system 800 also includes read only memory (ROM) and/or other static storage device 840 coupled to bus 810 to store static information and instructions for processor(s) 820.

[0048] Electronic system 800 can also be coupled via bus 810 to input/output (I/O) interface 850. I/O interface 850 provides an interface to I/O devices 855, which can include, for example, a cathode ray tube (CRT) or liquid crystal display (LCD), to display information to a computer user, an alphanumeric input device including alphanumeric and other keys and/or a cursor control device, such as a mouse, a trackball, or cursor direction keys. Electronic system 800 further includes network interface 860 to provide access to a network, such as a local area network, whether wired or wireless.

[0049] Instructions are provided to memory 835 from a storage device, such as magnetic disk, a read-only memory (ROM) integrated circuit, CD-ROM, DVD, via a remote connection (e.g., over a network via network interface 860) that is either wired or wireless, etc. In alternative embodiments, hard-wired circuitry can be used in place of or in combination with software instructions. Thus, execution of sequences of instructions is not limited to any specific combination of hardware circuitry and software instructions.

Conclusion

[0050] Reference in the specification to "one embodiment" or "an embodiment" means that a particular feature, structure, or characteristic described in connection with the embodiment is included in at least one embodiment of the invention. The

appearances of the phrase "in one embodiment" in various places in the specification are not necessarily all referring to the same embodiment.

[0051] In the foregoing specification, the invention has been described with reference to specific embodiments thereof. It will, however, be evident that various modifications and changes can be made thereto without departing from the broader spirit and scope of the invention. The specification and drawings are, accordingly, to be regarded in an illustrative rather than a restrictive sense.